PROBLEM 1:  \textit{(Drawing)}

A. Draw what \texttt{head} and \texttt{list} are pointing to after the following code is executed. Draw all of the list nodes and their contents.

\begin{verbatim}
String[] data = {"A", "B", "C", "D"};

ListNode head = new ListNode(data[0], null, null);
ListNode list = head;
for (int k=1; k < data.length; k++) {
    list.next = new ListNode(data[k], null, null);
    list = list.next;
}
head.prev = list;
\end{verbatim}

B. Draw what \texttt{root} and \texttt{next} points to after executing the following code. Indicate if there is some kind of error.

\begin{verbatim}
TreeNode root = new TreeNode(8, new TreeNode(3, new TreeNode(2, null, null),
                                         new TreeNode(7, null, null)),
                             new TreeNode(10, null, null));
TreeNode next = root.right.right;
\end{verbatim}
PROBLEM 2: (Counting trees (16 points))

The Counting BSTs APT asks you to create a class BSTcount that contains the method howMany, which takes an array of integers, and returns a long value that represents the number of distinct possible BSTs resulting from the given set of values.

A solution is given below that is syntactically correct, but does not currently solve any of the test cases.

```java
public class BSTcount {

    /**
     * Return the number of distinct possible BSTs resulting from the given values
     * @param values elements are distinct
     */
    public long howMany(int[] values) {
        return howMany(values.length);
    }

    /**
     * Return the number of distinct possible BSTs with size elements
     */
    public long howMany(int size) {
        long trees = 0;
        for (int left=0; left <= size-1; left++) {
            int right = size - 1 - left;

            trees += howMany(left)*howMany(right);
        }
        return trees;
    }

}
```
A. The solution above returns 0 for both values = {90,12} and values = {90,13,2,3} instead of the correct answers (2 and 14 respectively). Explain why and fix the code above. Your fix should be one or two lines of code.

B. After your fix, your program works but it is not able to complete in time for the APT grader for large arrays. Using memoization, you can avoid repeating the calculation of previous calls to howMany and significantly reduce the running time of your algorithm. Add code as necessary. You may use the space below if necessary.

C. Explain why the memoized solution is so much faster by stating the recurrences for the unmemoized and memoized versions. Do not solve the recurrence. You can use \ldots to indicate patterns like $n + (n - 1) + \ldots + 1$ is the sum of the sequence of numbers from $n$ to 1.
PROBLEM 3: (Grids (18 points))

Consider a $N$-by-$N$ grid in which some squares are occupied by black circles. Two squares belong to the same group if they share a common edge. In the picture to the right, there are
- 1 group of 4 occupied squares
- 1 group of 3 occupied squares
- 2 groups of 2 occupied squares
- 2 groups of individually occupied squares

Given that grid is an two-dimensional array where $grid[i][j] == true$ if and only if grid cell $(i, j)$ is occupied, the following questions ask you to find the groups.

Given a grid and a grid cell location $(row, col)$, $groupSize$ should compute the size of the group including that square. For example, in the example above $groupSize(grid, 1, 3)$ should return 2.

```java
public int groupSize(boolean[][] grid, int row, int col)
{
```
PROBLEM 4: (Puzzle Hunt)

You are given a matrix of positive integers to represent a game board, where the (0, 0) entry is the upper left corner. The number in each location is the number of squares you can advance in any of the four primary compass directions, provided that move does not take you off the board. You are interested in the total number of distinct ways one could travel from the upper left corner to the lower right corner, given the constraint that no single path should ever visit the same location twice.

Consider the initial game board to the left, and notice that the upper left corner is occupied by a 2. That means you can take either two steps to the right, or two steps down (but not two steps to the left or above, because that would carry you off the board). Suppose you opt to go right so that you find yourself in the configuration to the right.

After that, you could continue along as follows:

This series of moves illustrates just one of potentially several paths you could take from upper left to lower right. Your task is to write a method called numPaths, which takes a 2-d array of integers and computes the total number of ways to travel to the lower right corner of the board. Note that you never want to count the same path twice, but two paths are considered to be distinct even if they share a common sub-path. And because you want to prevent cycles, you should change the value at any given location to a zero as a way of marking that you’ve been there. Just be sure to restore the original value as you exit the recursive call.

You may want to write a helper function to handle the recursion and a utility function to decide if you are on the board or not.
A. Write `numPaths` below.

```java
/**
 * Calculates total number of distinct ways one could travel from the
 * upper left corner of grid to the lower right corner, given the
 * constraint that no single path should ever visit the same location twice.
 * @param board square matrix board[i][j] is the number of squares
 * one can advance vertically or horizontally from (i,j)
 * @return the number of possible paths from (0,0) to the lower
 * right corner of board (board.length-1, board[0].length - 1)
 */
public static int numPaths(int[][] board) {

}

// HELPER FUNCTIONS
/**
 * @return true if (row,col) is within the bounds of the board
 * (i.e. 0 <= row < board.length and 0 <= col < board[0].length)
 * false otherwise
 */
public static boolean onBoard(int[][] board, int row, int col) {

}

/**
 * @return the number of possible paths from (row,col) to the lower
 * right corner of board (board.length-1, board[0].length - 1)
 */
public static int numPaths(int[][] board, int row, int col) {

}
```

B. Give a recurrence for your solution. You do not need to solve the recurrence.